

Proyecto Final AA

Samuel Cardenete Rodríguez y Juan José Sierra González

2 de junio de 2017

Introducción:

El objetivo de esta práctica consistirá en realizar una serie de modelos y observar que cada uno ofrece una solución diferente en base a los datos de los que disponemos. Centraremos nuestro estudio en cuatro modelos no lineales, pero compararemos también junto a ellos el mejor modelo lineal que seamos capaces de estimar.

Para cada uno de los modelos trataremos de encontrar los parámetros que mejor lo ajustan para nuestro problema, y de explicar por qué cada uno se comporta de forma diferente. Como conclusión final nos quedaremos con el modelo que nos dé un mejor ajuste y explicaremos por qué.

“Housing Data Set”

En este caso nos encontramos frente a un problema de regresión, tal y como hemos visto anteriormente. Se trata de un conjunto de datos que estima el valor de distintas viviendas en los suburbios de Boston, basándose en algunas características como el crimen per capita, la accesibilidad a autopistas radiales, etc.

Como primer paso para abordar el problema, leeremos los datos y los dividiremos seleccionando nuestro conjunto de entrenamiento y de prueba.

Lectura de datos:

Procedemos a la lectura de la base de datos de regresión ‘Housing’.

```
datos_housing = as.data.frame( read_csv("datos/housing.data"))
```

```
## Parsed with column specification:
## cols(
##   CRIM = col_double(),
##   ZN = col_double(),
##   INDUS = col_double(),
##   CHAS = col_integer(),
##   NOX = col_double(),
##   RM = col_double(),
##   AGE = col_double(),
##   DIS = col_double(),
##   RAD = col_integer(),
##   TAX = col_double(),
##   PTRATIO = col_double(),
##   B = col_double(),
##   LSTAT = col_double(),
##   MEDV = col_double()
## )
```

```
head(datos_housing)
```

```
##      CRIM  ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD  TAX  PTRATIO      B
## 1  0.00632 18   2.31    0  0.538  6.575  65.2  4.0900   1  296    15.3 396.90
## 2  0.02731  0   7.07    0  0.469  6.421  78.9  4.9671   2  242    17.8 396.90
## 3  0.02729  0   7.07    0  0.469  7.185  61.1  4.9671   2  242    17.8 392.83
## 4  0.03237  0   2.18    0  0.458  6.998  45.8  6.0622   3  222    18.7 394.63
## 5  0.06905  0   2.18    0  0.458  7.147  54.2  6.0622   3  222    18.7 396.90
## 6  0.02985  0   2.18    0  0.458  6.430  58.7  6.0622   3  222    18.7 394.12
##      LSTAT  MEDV
## 1    4.98  24.0
```

```
## 2  9.14 21.6
## 3  4.03 34.7
## 4  2.94 33.4
## 5  5.33 36.2
## 6  5.21 28.7
```

Conjuntos de training y test usados

A continuación, realizaremos el particionamiento del conjunto de datos. Para el conjunto ‘train’ de entrenamiento emplearemos el 80% del conjunto total de los datos, de forma que el 20% restante será empleado para test. Hemos decidido utilizar este porcentaje dado que se nos ha explicado que un buen reparto entre train y test oscila entre dos terceras partes para train, o bien hasta un 80%, utilizando el resto para test.

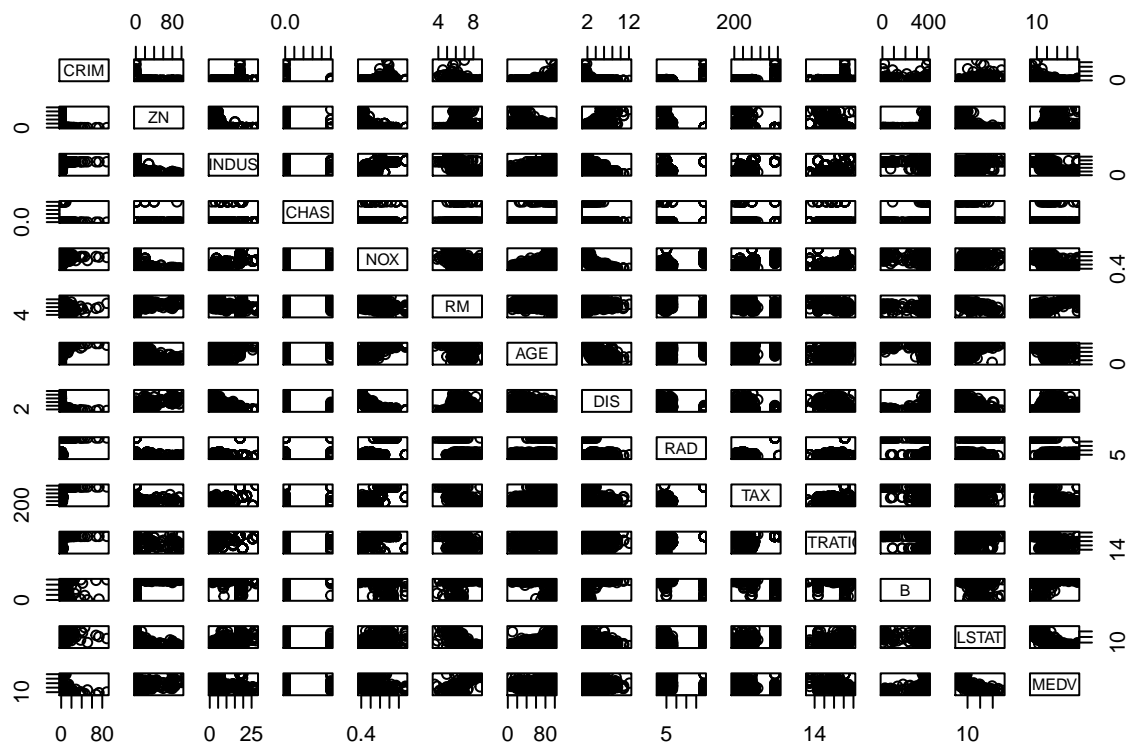
Procedemos al particionamiento de los datos, así como a su almacenamiento en respectivas variables:

```
train = sample(nrow(datos_housing), round(nrow(datos_housing)*0.8))
#definimos ambos conjuntos en dos data.frame diferentes:
housing_train = datos_housing[train,]
housing_test = datos_housing[-train,]
```

Preprocesamiento de los datos

Antes de entrar en el preprocesamiento, consideramos interesante realizar una vista preliminar de las gráficas de dispersión de cada atributo de la base de datos siendo reflejado con todos los demás. Así obtenemos una matriz simétrica de gráficas:

```
pairs(datos_housing)
```



Al representar la matriz de diagramas podemos hacernos una primera idea de la dispersión de los datos. Como podemos observar esta dispersión es alta, pero la columna que más nos interesa sobre todo es la de la variable respuesta, ‘MEDV’. En este caso podemos observar que la variable ‘LSTAT’ parece presentar una

fuerte representatividad con respecto a la variable respuesta, la podemos marcar a priori como una a tener en cuenta a la hora de realizar modelos.

En este caso, estamos trabajando sobre un conjunto de datos que posee 14 atributos, para realizar una reducción de datos debemos de estar seguros de que existen atributos redundantes para el aprendizaje de nuestros modelos.

A primera vista la variabilidad de los datos es alta, no parece haber ningún atributo redundante, veamos entonces si es aconsejable realizar una reducción de la dimensionalidad de los datos mediante la aplicación de PCA (Computing the Principal Components) sobre el conjunto de datos. Estos atributos redundantes podrían ser recombinados en nuevas características como combinaciones lineales de ellos.

PCA calcula la varianza de cada atributo respecto a los demás, de forma que aquellos atributos que posean menor varianza (cercana a cero) con respecto a los demás serán considerados como redundantes.

Además, para no arriesgar mucho (puesto que PCA trabaja “a ciegas”) representaremos sólo aquellos componentes que expliquen hasta el 90% de la variabilidad de los datos (es necesario que los datos estén escalados y centrados para aplicar PCA):

```
housingTrans = preprocess(datos_housing[, -which(names(datos_housing) == "MEDV")], method = c("BoxCox",
housingTrans$rotation
```

##		PC1	PC2	PC3	PC4	PC5
##	CRIM	0.351358922	-0.080323795	0.21229244	-0.03046747	0.033900319
##	ZN	-0.255408294	0.026705781	0.40186216	-0.13992202	-0.300421126
##	INDUS	0.337559710	-0.035840397	-0.09365446	-0.03972878	0.033764630
##	CHAS	0.008802168	-0.567171494	-0.23494015	-0.75574795	-0.160554801
##	NOX	0.345753987	-0.204606754	-0.08234027	0.18010421	-0.014959856
##	RM	-0.183045703	-0.509627274	0.33596340	0.25887613	0.403709095
##	AGE	0.307325638	-0.196513392	-0.22455474	0.24264930	0.001885428
##	DIS	-0.326395109	0.246224763	0.13552775	-0.22911923	-0.020886542
##	RAD	0.294095534	-0.032632643	0.42351142	-0.20415425	0.178466288
##	TAX	0.305780705	0.035027906	0.38197798	-0.16442219	0.059603078
##	PTRATIO	0.197381672	0.426164494	0.02183027	-0.33884117	0.523471973
##	B	-0.195668297	0.002287144	-0.41362881	-0.10328662	0.556939871
##	LSTAT	0.293283963	0.291762870	-0.21021966	-0.04167087	-0.313731923
##		PC6	PC7			
##	CRIM	-0.08661682	0.1034686268			
##	ZN	-0.42440659	-0.5480987095			
##	INDUS	-0.01982528	0.1706661615			
##	CHAS	0.12777740	-0.0629487572			
##	NOX	-0.09608618	0.0004566907			
##	RM	0.22368487	-0.2592036155			
##	AGE	-0.03726619	-0.5444595484			
##	DIS	0.09479850	0.0624644483			
##	RAD	-0.17887309	0.2017656634			
##	TAX	-0.29408743	0.0408997952			
##	PTRATIO	0.39295495	-0.3966467824			
##	B	-0.66903319	-0.0022056976			
##	LSTAT	-0.07423922	-0.2975770855			

Como podemos observar en la tabla, se han reducido los atributos a 7 atributos (combinaciones lineales de los 14 anteriores). Pero si observamos las varianzas de cada atributo en la tabla respecto al resto de atributos, vemos que no existe ningún atributo cuyas varianzas sean cercanas todas a 0. Aún así, para asegurarnos lo comprobamos mediante la siguiente función:

```
nearZeroVar(housingTrans$rotation)
```

```
## integer(0)
```

Como comprobamos con la función `nearZeroVar` no existe ningún atributo cuyas varianzas respecto a las demás sean todas cercanas a 0, y por tanto todos los atributos son considerados representativos, pues poseen dispersión. En conclusión, no realizaremos ninguna reducción de atributos.

Para concluir el preprocesamiento de los datos, realizaremos las siguientes operaciones sobre ellos:

- **Escalado:** Se trata de dividir cada uno de los atributos por su desviación típica.
- **Centrado:** Calculamos la media de cada atributo y se la restamos para cada valor.
- **Box-Cox:** Se trata de intentar reducir el sesgo de cada atributo, para intentar hacer este más próximo a una distribución Gaussiana.

Para aplicar las transformaciones, emplearemos la función `preProcess` sobre nuestro conjunto `train`, de forma que realizando un predict sobre el objeto transformación obtenido, obtengamos el conjunto de datos `train` preprocesado. A continuación, realizaremos **las mismas transformaciones** sobre el conjunto de `test`, utilizando el mismo objeto transformación:

```
housingTrans = preProcess(housing_train[, -which(names(housing_train) == "MEDV")],
                          method = c("BoxCox", "center", "scale"), thresh = 0.9)
```

```
housing_train[, -which(names(housing_train) == "MEDV")] = predict(housingTrans,
                                                                housing_train[, -which(names(housing_train) == "MEDV")])
```

```
housing_test[, -which(names(housing_test) == "MEDV")] = predict(housingTrans, housing_test[, -which(names(housing_test) == "MEDV")])
```

Estimación de parámetros

Antes de realizar un modelo, veamos cuáles son las características más representativas (las que ofrecen varianza mayor con respecto al resto de datos), de forma que no empecemos a realizar modelos a ciegas, sino fijándonos en la calidad de sus atributos.

Para ello emplearemos la función `regsubsets`. Esta función realiza una búsqueda exhaustiva (empleando Branch&Bound) de las mejores agrupaciones de atributos en nuestro conjunto de entrenamiento para predecir en una regresión lineal (será el primer modelo a realizar):

```
regsub_housing = regsubsets(datos_housing[, -which(names(datos_housing) == "MEDV")],
                           datos_housing[, which(names(datos_housing) == "MEDV")])
```

```
summary(regsub_housing)
```

```
## Subset selection object
## 13 Variables (and intercept)
##           Forced in Forced out
## CRIM          FALSE      FALSE
## ZN            FALSE      FALSE
## INDUS         FALSE      FALSE
## CHAS          FALSE      FALSE
## NOX           FALSE      FALSE
## RM           FALSE      FALSE
## AGE          FALSE      FALSE
## DIS          FALSE      FALSE
## RAD          FALSE      FALSE
## TAX          FALSE      FALSE
```

```
## PTRATIO      FALSE      FALSE
## B            FALSE      FALSE
## LSTAT        FALSE      FALSE
## 1 subsets of each size up to 8
## Selection Algorithm: exhaustive
##           CRIM ZN  INDUS CHAS NOX RM  AGE DIS RAD TAX PTRATIO B  LSTAT
## 1  ( 1 ) " "  " " " " " " " " " " " " " " " " " " " " " " " " " " " "
## 2  ( 1 ) " "  " " " " " " " " " "*" " " " " " " " " " " " " " " " " "
## 3  ( 1 ) " "  " " " " " " " " " "*" " " " " " " " " " " " " " " " " "
## 4  ( 1 ) " "  " " " " " " " " " "*" " " " " "*" " " " " " " " " " " "
## 5  ( 1 ) " "  " " " " " " " " " "*" "*" " " " "*" " " " " " " " " " " "
## 6  ( 1 ) " "  " " " " " " "*" "*" "*" " " " "*" " " " " " " " " " " "
## 7  ( 1 ) " "  " " " " " " "*" "*" "*" " " " "*" " " " " " " " " " " "
## 8  ( 1 ) " "  "*" " " " " "*" "*" "*" " " " "*" " " " " " " " " " " "

```

En la gráfica aportada por la función, obtenemos los atributos más representativos en caso de realizar un modelo con ‘n’ parámetros (filas), de forma que según el número de atributos que deseemos para nuestro modelo nos estima cuales serían los parámetros más representativos a emplear. Nuestro criterio a seguir a la hora de generar modelos lineales será tratar de predecir utilizando las agrupaciones de atributos que la función considera representativos, es decir, si queremos realizar un modelo con n características, utilizaremos aquellas que estén marcadas con una estrella para la fila n. Para mostrar un ejemplo que explique la gráfica, si queremos utilizar 3 atributos para el modelo, escogeremos ‘LSTAT’, ‘RM’ y ‘PTRATIO’.

Ahora que sabemos cómo elegir las características más recomendables para realizar modelos, podremos construir una serie de modelos lineales con algunas de estas características y validar con el conjunto de test para comprobar los errores que reflejan. Luego compararemos los resultados del mejor modelo obtenido con el resto de modelos no lineales que son objeto de estudio del proyecto.

Regularización

Antes de generar cualquier modelo también es importante realizar un análisis para ver si es interesante aplicar una regularización empleando Weight-decay mediante la función glmnet. Dicha función recibe los siguientes hiperparámetros:

- **Alpha:** Para aplicar el weight-decay utilizaremos dicho argumento con valor 0.
- **Lambda:** Parámetro de regularización. (Multiplica la matriz de identidad)

Hace falta tener en cuenta antes la correcta elección del lambda, de forma que escojamos el que mejores resultados nos pueda arrojar. En lugar de seleccionarlo de forma arbitraria, será mejor emplear validación cruzada:

```
etiquetas = housing_train[,which(names(housing_train) == "MEDV")]
tr = housing_train[,-which(names(housing_train) == "MEDV")]
tr = as.matrix(tr)
crossvalidation = cv.glmnet(tr,etiquetas,alpha=0)
print(crossvalidation$lambda.min)
```

```
## [1] 0.7884988
```

Una vez obtenido el lambda que proporciona un menor E_{out} , procedemos a generar un modelo de regularización, en primer lugar empleando el valor de lambda generado por validación cruzada, y en segundo lugar empleando un lambda igual a cero, de forma que no apliquemos regularización. El objetivo será comprobar si los parámetros obtenidos son significativamente diferentes como para que merezca la pena realizar regularización en nuestros modelos.

```
modelo_reg = glmnet(tr,etiquetas,alpha=0,lambda=crossvalidation$lambda.min)
print(modelo_reg)
```

```
##
## Call:  glmnet(x = tr, y = etiquetas, alpha = 0, lambda = crossvalidation$lambda.min)
##
##      Df    %Dev Lambda
## [1,] 13 0.7692 0.7885
```

Aplicando regularización con el lambda obtenido tras la validación cruzada obtenemos una desviación del 0.769. Probemos ahora no aplicando regularización, empleando un hiperparámetro lambda de 0, y comprobemos su desviación:

```
modelo_reg = glmnet(tr,etiquetas,alpha=0,lambda=0)
print(modelo_reg)
```

```
##
## Call:  glmnet(x = tr, y = etiquetas, alpha = 0, lambda = 0)
##
##      Df    %Dev Lambda
## [1,] 13 0.7779      0
```

Como podemos comprobar, las desviaciones obtenidas empleando o no regularización mediante weight-decay son similares (0.769 frente a 0.778), por tanto, concluimos en que emplear regularización no merece la pena.

Definición de modelos lineales

Para la realización de los modelos planteamos emplear regresión logística mediante la función ‘glm’; pero si empleamos regresión logística obtendremos un conjunto de probabilidades que predigan los valores. Para ello, como precondition sería necesario normalizar los valores de la variable respuesta de forma que se encontraran en el intervalo [0,1].

Entonces procederemos a generar modelos mediante regresión lineal, así obtendremos un valor dependiendo de la distancia de los atributos a la recta generada.

```
m_muestra_housing = lm(MEDV ~ LSTAT, data=housing_train)
```

Antes definimos una función para el cálculo del error. Para ello realizamos un cálculo de la media de errores cuadráticos. Utilizamos la función *mse*, a la que llamamos una vez que se han calculado las probabilidades en base a un modelo.

Sean Y las etiquetas verdaderas e Y' las predichas por nuestro modelo:

$$ECM = \frac{1}{n} \sum_{i=1}^n (Y'_i - Y_i)^2$$

```
calculoMSE = function (modelo, test, variable_respuesta){
  prob_test = predict(modelo, test[,which(names(test) == variable_respuesta)])

  mse(test[,which(names(test) == variable_respuesta)], prob_test)
}
```

Utilizamos nuestra función para calcular el error del modelo de muestra generado anteriormente:

```
etest_mmuestra = calculoMSE(m_muestra_housing, housing_test, "MEDV")
etest_mmuestra
```

```
## [1] 24.58877
```

Obtenemos un error de 24.59, pero se trata de un modelo excesivamente simple como para reflejar buenos resultados en un problema real. Por tanto busquemos un modelo diferente empleando otra característica, la siguiente más representativa en el conjunto de datos:

```
m1_housing = lm(MEDV ~ LSTAT + RM, data=housing_train)

etest_m1 = calculoMSE(m1_housing, housing_test, "MEDV")
etest_m1
```

```
## [1] 25.34943
```

Utilizando dos características el error en el conjunto de test aumenta. No podemos asegurar que se haya producido sobreajuste para tan solo dos características, así que seguimos probando nuevos modelos, añadiendo la siguiente característica recomendada por regsubsets:

```
m2_housing = lm(MEDV ~ LSTAT + RM + PTRATIO, data=housing_train)

etest_m2 = calculoMSE(m2_housing, housing_test, "MEDV")
etest_m2
```

```
## [1] 23.62471
```

El error ya sí ha descendido, por lo que nos vemos capacitados de seguir probando con nuevas características. Como aún nos queda una buena cantidad por probar, añadimos una más al siguiente modelo:

```
m3_housing = lm(MEDV ~ LSTAT + RM + PTRATIO + DIS, data=housing_train)

etest_m3 = calculoMSE(m3_housing, housing_test, "MEDV")
etest_m3
```

```
## [1] 19.90819
```

En esta ocasión tenemos un error de 19.908, que mejora sustancialmente al que teníamos antes. Probemos con la siguiente característica:

```
m4_housing = lm(MEDV ~ LSTAT + RM + PTRATIO + DIS + NOX, data=housing_train)

etest_m4 = calculoMSE(m4_housing, housing_test, "MEDV")
etest_m4
```

```
## [1] 20.24523
```

En este momento ya se produce sobreajuste, por lo que pasamos a probar con transformaciones no lineales sobre las características que han presentado el mejor modelo hasta ahora:

```
m5_housing = lm(MEDV ~ LSTAT + RM + I(PTRATIO^2) + DIS, data=housing_train)

etest_m5 = calculoMSE(m5_housing, housing_test, "MEDV")
etest_m5
```

```
## [1] 20.28286
```

Hemos probado a realizar una transformación cuadrática sobre uno de los atributos, y el error no ha reducido. La última transformación que probaremos será multiplicar las componentes del modelo entre sí:

```
m6_housing = lm(MEDV ~ LSTAT * RM * PTRATIO * DIS, data=housing_train)

etest_m6 = calculoMSE(m6_housing, housing_test, "MEDV")
etest_m6
```



```
## [1] 11.10787
```

Con este modelo lineal con transformaciones no lineales hemos alcanzado un error de 11.108, mejor que los anteriores. Por tanto, nos quedamos con este modelo como el mejor lineal. ¿Es este mejor que un modelo no lineal? A continuación probaremos distintos modelos no lineales y compararemos los resultados para ver cuál ajusta mejor.

MODELOS NO LINEALES:

Definiremos cuatro modelos no lineales con los que comparar el mejor modelo lineal que hemos encontrado y poder realizar una valoración objetiva de cuál ofrece mejores resultados para nuestra base de datos. Los modelos a estudiar serán:

- **Redes Neuronales**
- **Support Vector Machine**
- **Boosting**
- **Random Forest**

REDES NEURONALES:

Comenzaremos abordando las redes neuronales. Para realizar modelos de redes neuronales utilizaremos el paquete “neuralnet”, que nos aporta una función para el estudio de estos modelos.

Para la estimación de los parámetros a obtener, tal como se especifica en el guión, emplearemos validación cruzada para estimar el E_{out} . En nuestro caso vamos a dividir nuestro conjunto de entrenamiento en 5 particiones y en cada iteración validaremos con una habiendo entrenado con las 4 restantes. Esto se denomina 5-fold-cross-validation.

Hemos diseñado una función que realiza los cálculos pertinentes para un conjunto de datos dado y una arquitectura de la red neuronal. El objetivo será llamar a la función con distintas arquitecturas con 1, 2 y 3 capas ocultas y con distintas unidades en cada una de ellas, y finalmente encontrar cuál ofrece un error más reducido. Los pesos se inicializarán a valores aleatorios siguiendo una distribución normal si no se especifica un vector inicial.

En primer lugar, para que la red neuronal tenga éxito, escalaremos y centraremos los datos independientemente de las transformaciones realizadas anteriormente. El objetivo es normalizar los valores, incluyendo la variable respuesta en este caso. Con estos datos podremos construir nuestra red neuronal, pero a continuación será necesario “desnormalizar” los valores obtenidos para que el error sea comparable al resto de errores obtenidos en la práctica. [1]

```
crossValidationNN = function(datos, capas){  
  maxs = apply(datos, 2, max)  
  mins = apply(datos, 2, min)  
  datos = as.data.frame(scale(datos, center = mins, scale = maxs - mins))  
  set.seed(6)  
  #realizamos 5 particiones:  
  folds = suppressWarnings(split(datos, sample(rep(1:5, nrow(datos)/5))))  
  errores = as.numeric()  
  for (i in 1:5){  
    test_ = folds[[i]]  
    train_ = data.frame()  
    for (j in 1:5){  
      if(j!=i)  
        train_ = rbind(train_, folds[[j]])  
    }  
  }
```

```

n = names(train_)
f = as.formula(paste("MEDV ~", paste(n[!n %in% "MEDV"], collapse = " + ")))
nn = neuralnet(f,data=train_,hidden=capas,linear.output=T)
pr.nn = compute(nn,test_[,1:13])
pr.nn_ = pr.nn$net.result*(max(datos_housing$MEDV)-min(datos_housing$MEDV))+min(datos_housing$MEDV)
test.r = (test_$MEDV)*(max(datos_housing$MEDV)-min(datos_housing$MEDV))+min(datos_housing$MEDV)
MSE.nn = sum((test.r - pr.nn_)^2)/nrow(test_)

errores = c(errores, MSE.nn)
}

mean(errores)
}

eout_nn_1_5=crossValidationNN(datos_housing,5)
eout_nn_1_5

## [1] 17.53102175

eout_nn_1_3=crossValidationNN(datos_housing,3)
eout_nn_1_3

## [1] 22.21987699

eout_nn_2_53=crossValidationNN(datos_housing,c(5,3))
eout_nn_2_53

## [1] 12.63083018

eout_nn_2_74=crossValidationNN(datos_housing,c(7,4))
eout_nn_2_74

## [1] 16.35025809

eout_nn_3_853=crossValidationNN(datos_housing,c(8,5,3))
eout_nn_3_853

## [1] 12.55428639

eout_nn_3_742=crossValidationNN(datos_housing,c(7,4,2))
eout_nn_3_742

## [1] 11.30877295

```

Hemos realizado pruebas con 6 arquitecturas, 2 con 1 capa interna, 2 con 2 y otras 2 con 3. Las unidades por capa han sido elegidos arbitrariamente, ya que el objetivo en este caso es comparar redes neuronales, y a base de cross-validation y pruebas con distintos valores encontrar qué arquitectura con qué número de unidades por capa parece ser la mejor.

En nuestro caso, vemos que el menor error viene obtenido con la última de las redes, la que tiene 3 capas con 7, 4 y 2 unidades por capa respectivamente. Sin embargo, también hay una arquitectura de 2 capas que ofrece un error un poco por encima, pero que también podemos considerar aceptable. Siguiendo el criterio de la navaja de Ockham, para dos soluciones prácticamente iguales, la mejor solución es aquella que resuelve el problema de forma más simple.

Por tanto, podríamos quedarnos con cualquiera de las dos redes, ya que consideramos que ambas pueden considerarse una buena solución en término medio de eficiencia/error.

Por último, ofrecemos una gráfica de cómo se ha generado la mejor red neuronal y qué valores toman los pesos y el sesgo en cada una de las capas para un caso puntual.

```
maxs = apply(datos_housing, 2, max)
mins = apply(datos_housing, 2, min)
datos_escalados = as.data.frame(scale(datos_housing, center = mins, scale = maxs - mins))
housing_train_esc = datos_escalados[train,]
n = names(housing_train_esc)
f = as.formula(paste("MEDV ~", paste(n[!n %in% "MEDV"], collapse = " + ")))
nn_winner = neuralnet(f, data=housing_train_esc, hidden=c(7,4,2), linear.output=T)
plot(nn_winner)
```

Debido a un error que no hemos sido capaces de solucionar, la función plot no nos muestra en este caso la figura que debe mostrar en el PDF. Sin embargo, como al ejecutar el código R sí muestra una gráfica, se la adjuntamos en “Anexo1.png” dentro del fichero zip donde entregamos la práctica.

SUPPORT VECTOR MACHINE

El siguiente modelo a tratar será SVM (Support Vector Machine). Este modelo trata de estimar un clasificador que pueda separar los datos con el máximo margen, para que nuevos datos que puedan oscilar entre los de la muestra se aproximen bien de igual forma.

Para estimar el mejor valor para un parámetro podemos utilizar la función tune. Esta función toma una serie de parámetros con los que experimentar y un rango y un salto para cada uno de ellos. La idea es que se estime el valor de cada parámetro que hace que el error en un 10-fold-cross-validation sea menor. Para ello esta función recibe un modelo, las características con las que predecir la variable respuesta, el conjunto de datos y la lista de parámetros a estimar.

Tal y como se nos indica en el enunciado, nosotros trataremos de estimar el valor del parámetro libre gamma. Como se nos pide una precisión de dos cifras, solicitamos a la función que pruebe todos los valores entre 0 y 1 con un salto de 0.01.

```
tune_gamma_svm = tune(svm, MEDV ~ ., data=housing_train,
                      ranges = list(gamma=seq(0,1,0.01)))

tune_gamma_svm
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   gamma
##   0.05
##
## - best performance: 16.09606119
```

Tras la ejecución de la función obtenemos que 0.09 es el mejor valor para gamma que ha encontrado. A continuación procederemos a realizar un estudio con cross-validation sobre el conjunto completo de datos utilizando este parámetro. Definimos una nueva función similar a la de las redes neuronales.

El proceso de partición del conjunto de datos es igual al anterior. Ahora para cada par train-test entrenaremos el modelo con el conjunto de train con el parámetro gamma tomando el valor anteriormente calculado. Utilizamos el núcleo gaussiano pasándole el parámetro “radial” como kernel. El error nuevamente es el error medio cuadrático.

```

crossValidationSVM = function(datos){
  set.seed(6)
  #realizamos 5 particiones:
  folds = suppressWarnings(split(datos, sample(rep(1:5, nrow(datos)/5))))
  errores = as.numeric()
  for (i in 1:5){
    test_ = folds[[i]]
    train_ = data.frame()
    for (j in 1:5){
      if(j!=i)
        train_ = rbind(train_, folds[[j]])
    }

    svm_housing = svm(MEDV ~ ., data=train_, kernel="radial", gamma=0.09)
    pred_svm = predict(svm_housing, test_)
    error = mse(test_$MEDV, pred_svm)

    errores = c(errores, error)
  }

  mean(errores)
}

crossValidationSVM(datos_housing)

```

```
## [1] 15.20308769
```

Como podemos observar, el error que nos da SVM está por encima de los otros dos que hemos generado hasta el momento, el del modelo lineal y el de la red neuronal. SVM es un algoritmo que funciona principalmente bien en problemas de clasificación binaria, ya que su comportamiento tratando de dejar el mayor margen entre los datos y el clasificador proporciona unas zonas muy óptimas a la hora de tratar nuevos datos.

Sin embargo, en un problema de regresión como el que nos atañe esa particularidad tiende a reflejar un error mayor, pues la distancia entre los datos y el clasificador es justo lo que tratamos de optimizar. Por este motivo consideramos entendible que un modelo como SVM refleje malos resultados en este tipo de problema.

BOOSTING

A continuación pasamos a valorar el modelo de Boosting. Como hemos aprendido, Boosting es un método muy potente que basa su clasificador final en una serie de clasificadores simples previos. El final será una combinación de los anteriores que separe de forma correcta o próxima los datos.

Antes de generar el modelo, vamos a realizar una estimación (tuning) de los parámetros más interesantes a nuestro parecer para Boosting. Estos son:

- **n.trees**: el número de árbol generados, el cual estimaremos entre 50 y 500.
- **interaction.depth**: la profundidad de cada árbol, es decir, su complejidad. La estimamos entre 2 y 10.
- **shrinkage**: la tasa de aprendizaje, la cual vamos a estimar entre 0.01 y 0.1, que son dos valores de tasa de aprendizaje que hemos utilizado en prácticas anteriores, para ver cuál se ajusta más a nuestro problema.

Vamos a realizar una rejilla de valores que deben tomar los distintos parámetros para cada una de las ejecuciones, de forma que se pueda estimar la mejor. La construiremos utilizando la función `expand.grid`, añadiendo como columnas los nombres de los atributos y como filas las combinaciones de valores a tomar.

Además, hemos tenido que añadir una columna con los valores de `n.minobsinnode` porque lo requería la función. En este caso vamos a dejarle el valor que toma por defecto según la documentación de `gbm`, que es 10.

Nuestra rejilla tendrá entonces 50 combinaciones, aquellas posibles entre los 10 valores diferentes que tomará `n.trees` y los 5 que tomará `interaction.depth`.

Para llevar a cabo cada una de las experimentaciones con los modelos, utilizaremos la técnica de Bagging, generando conjuntos mediante Bootstrap con los que poder validar cada modelo un número considerable de veces y obtener un valor de error significativo que nos permita determinar con certeza el valor idóneo de los parámetros (aquellos valores que presenta el mejor modelo generado).

El número de conjuntos generados con Bootstrap será indicado en el objeto `bootstrap_control`, generado mediante la función `trainControl` y a la que le pasamos el número de conjuntos a ser empleados. La técnica con la que se seleccionarán estos conjuntos se especificará a continuación en la función `train`, del paquete “caret”.

En la llamada a esta función se pueden especificar multitud de parámetros, pero los que nosotros vamos a especificar son los datos y la variable respuesta, lógicamente, y además el tipo de modelo a entrenar (`gbm = Boosting`), el número de conjuntos a generar por Bootstrapping (con nuestro objeto `bootstrap_control`), indicando el método especificando que la variable `bag.fraction` sea 1 (cada uno de los conjuntos se generan de igual tamaño al conjunto de datos, con reemplazamiento), y por último, la rejilla que contiene los parámetros a estimar, que es el grid que hemos creado anteriormente. [2]

```
grid_gbm = expand.grid(.interaction.depth = (1:5)*2, .n.trees = (1:10)*50, .shrinkage = 0.1, .n.minobsinnode = 10)

bootstrap_control = trainControl(number = 200)
gbm_housing = train(housing_train[,1:13], housing_train[,14],
                    method="gbm", trControl = bootstrap_control,
                    verbose=F, bag.fraction = 1,
                    tuneGrid = grid_gbm)

gbm_housing$bestTune

##      n.trees interaction.depth shrinkage n.minobsinnode
## 17      350                4        0.1              10
```

Una vez hemos terminado de estimar los parámetros, podemos acceder a la mejor combinación utilizando la componente `bestTune` del objeto construido. Vemos que nuestra mejor estimación supone utilizar 350 árboles con una profundidad de 4.

Ahora procedamos a validar dicho modelo empleando 5fold cross-validation con los parámetros obtenidos. Analizaremos el error medio cuadrático obtenido (E_{out}) como siempre con la función `mse`.

```
set.seed(7)
boosting_housing = gbm(MEDV ~ ., data = housing_train,
                      n.trees = 350, interaction.depth = 4,
                      shrinkage = 0.1, n.minobsinnode = 10,
                      cv.folds = 5)

## Distribution not specified, assuming gaussian ...

pred_boosting = predict(boosting_housing, housing_test,
                      n.trees = 350, interaction.depth = 4,
                      shrinkage = 0.1,
                      n.minobsinnode = 10)

mse(housing_test$MEDV, pred_boosting)
```

```
## [1] 7.201165429
```

El error que hemos obtenido habiendo optimizado los parámetros es el mejor que hemos encontrado hasta ahora, recompensando el trabajo invertido en encontrar aquellos valores que parecían minimizar el error en nuestro experimento previo.

El clasificador obtenido resulta de una combinación de clasificadores simples, que van clasificando los datos en función de unos pesos. Estos pesos se inicializan a un valor equitativo y se van modificando en función de si cómo de bien ha sido determinada su variable respuesta. Si un dato ha obtenido una mala valoración de su variable respuesta una gran cantidad de veces, su peso se incrementará para priorizar que se aproxime bien en próximas iteraciones. Del mismo modo, un dato bien aproximado con asiduidad verá su peso reducido para no ser tan tenido en cuenta a la hora de generar nuevos clasificadores.

El resultado es que se consigue que todos o la mayoría de los datos estén bien aproximados por algún clasificador simple y se construye un clasificador complejo que utiliza distintas secciones de los clasificadores previos. En un conjunto de datos como el nuestro se ha comprobado experimentalmente que los datos se acercan al clasificador bastante más que en otros modelos.

RANDOM FOREST

El último modelo no lineal a comparar será Random Forest. Este modelo funciona de forma similar a Bagging, pero con una serie de árboles no correlados al no utilizar para cada caso todos los predictores posibles. En particular, para problemas de regresión, en la documentación del paquete `randomForest` se puede encontrar que el valor ideal de 'm' (las variables para ramificaciones) es $p/3$, siendo p el número total de predictores.

Es un algoritmo que funciona de forma muy eficiente gracias a generar árboles con menos predictores, y por tanto menor profundidad. En el caso de los árboles, su complejidad es cuadrática por lo que a mayor número de predictores, el tiempo de cómputo crece exponencialmente. Es decir, es mucho más eficiente resolver muchos árboles pequeños que unos pocos grandes.

En primer lugar, como se indica en la práctica, debemos valorar el número de árboles a generar que ofrece un mejor resultado experimentalmente. Para ello utilizaremos de nuevo la función `tune`, cuyo objetivo será repetir el modelo Random Forest para todos los valores entre 25 y 500, en saltos de 25. Dado que Random Forest es un modelo que reduce el error muy rápidamente con los primeros árboles como hemos visto en teoría no hemos querido probar con valores más altos.

```
set.seed(6)
tune_rf = tune(randomForest, MEDV ~ ., data=housing_train,
               ranges = list(ntree=seq(25,500,25)))

tune_rf
```

```
##
## Parameter tuning of 'randomForest':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   ntree
##     50
##
## - best performance: 10.85101733
```

Esta ejecución nos refleja que el mejor valor indicado para el número de árboles es 50, habiendo realizado la cross-validation con nuestro conjunto de train. Ahora, procederemos a validar este modelo construyendo una función similar a la que utilizamos en redes neuronales que realice nuestra propia cross-validation sobre el conjunto completo de los datos. Este error, medido de nuevo con mse, nos reflejará un valor significativo.

```

crossValidationRF = function(datos){
  set.seed(6)
  #realizamos 5 particiones:
  folds = suppressWarnings(split(datos, sample(rep(1:5, nrow(datos)/5))))
  errores = as.numeric()
  m = ncol(housing_train)/3
  for (i in 1:5){
    test_ = folds[[i]]
    train_ = data.frame()
    for (j in 1:5){
      if(j!=i)
        train_ = rbind(train_, folds[[j]])
    }

    rf_housing = randomForest(MEDV ~ ., train_, ntree=50, mtry=m)
    error = calculoMSE(rf_housing, test_, "MEDV")

    errores = c(errores, error)
  }

  mean(errores)
}

eout_rf = crossValidationRF(datos_housing)
eout_rf

```

```
## [1] 10.46621588
```

Podemos comprobar que el error que nos presenta este modelo de Random Forest es el segundo mejor que hemos tenido hasta ahora, y en un tiempo escaso dado a la eficiencia del algoritmo. Como hemos indicado antes, Random Forest utiliza únicamente una fracción de los predictores en cada árbol, por lo que es posible que haya predictores buenos que se utilicen poco o no se utilicen, y desde luego el árbol final no tiene por qué seleccionar los predictores de forma óptima dado que los árboles a partir de los cuáles se genera no tienen por qué serlo.

Con esto, en resumen, podemos entender que este algoritmo no nos dé la mejor solución pero sí una muy buena en relación eficiencia/error. Y seguramente será uno de los principales algoritmos a tener en cuenta cuando contamos con una base de datos sin muchos predictores ya que es menos probable que se quede un número importante sin utilizar o sin dar el máximo que ofrecen.

Conclusión y modelo final seleccionado

Una vez planteados todos los modelos, habiendo realizado una validación cruzada para estimar un valor de error significativo para cada uno, y habiendo sido comparados, **Boosting** ha sido el que ha reflejado un menor error.

Debido a la forma de trabajar de Boosting, utilizando clasificadores simples para obtener un clasificador global que utilice secciones de cada uno, el clasificador obtenido fue muy bueno. Además es mucho más eficiente que otros modelos como las redes neuronales que se aprovechan más de la alta cantidad de características. Partiendo de “sólo” 14 como tenemos en este problema, y con además pocas muestras, no cualquier modelo funciona igual de bien.

Y justo eso es uno de los principales motivos por el que este algoritmo es uno de los más a tener en cuenta cuando contamos con una base de datos sin muchas muestras, porque nos favorece su planteamiento de remuestreo con reemplazamiento. El empleo del Bagging nos permite explotar bien los pocos datos de los que

disponemos y así mediante Bootstrapping ser capaces de entrenar y validar distintos conjuntos que nos lleven a un buen ajuste.

Concluimos recordando que, además, en este caso en particular se incidió en la optimización de dos parámetros visiblemente importantes, como son el número de árboles a generar por modelo y la profundidad de esos árboles. Habiendo podido explorar tanta variedad de modelos, el que obtengamos como el mejor podemos estar seguro de que será muy válido para nuestros datos. Y, experimentalmente, hemos podido demostrar que así es, siendo con un amplio margen el que obtiene el mejor ajuste.

Bibliografía

- [1] Fitting a Neural Network in R; neuralnet package, Datascience+, <https://datascienceplus.com/fitting-neural-network-in-r/>
- [2] Building Predictive Models in R Using the caret Package, Max Kuhn, Pfizer Global R&D, 2008.